2010

Researches On Reverse Lookup Problem In Distributed File System

Junyao Zhang
University of Central Florida

Researches on Reverse Lookup problem in Distributed File System

by

Junyao Zhang
B.S. Jilin University, 2007
M.S. Jilin University, 2009

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the School of Electronically Engineering and Computer Science
in the College of Electrical Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2010

Major Professor:
Jun Wang

Abstract

Recent years have witnessed an increasing demand for super data clusters. The super data clusters have reached the petabyte-scale can consist of thousands or tens of thousands storage nodes at a single site. For this architecture, reliability is becoming a great concern. In order to achieve a high reliability, data recovery and node reconstruction is a must. Although extensive research works have investigated how to sustain high performance and high reliability in case of node failures at large scale, a reverse lookup problem, namely finding the objects list for the failed node remains open. This is especially true for storage systems with high requirement of data integrity and availability, such as scientific research data clusters and etc. Existing solutions are either time consuming or expensive. Meanwhile, replication based block placement can be used to realize fast reverse lookup. However, they are designed for centralized, small-scale storage architectures. In this thesis, we propose a fast and efficient reverse lookup scheme named Group-based Shifted Declustering (G-SD) layout that is able to locate the whole content of the failed node. G-SD extends our previous shifted declustering layout and applies to large-scale file systems. Our mathematical proofs and real-life experiments show that G-SD is a scalable reverse lookup scheme that is up to one order of magnitude faster than existing schemes.

To my parents, to whom I owe so much

Acknowledgments

This thesis would not have been possible without the help and support of a number of people. First and foremost, I would like to express my sincerest gratitude to my advisor, Dr. Jun Wang, for the tremendous time, energy and wisdom he invested in my graduate education. His inspiring and constructive supervision has always been a constant source of encouragement for my study. I also want to thank my other thesis committee members, Dr. Shaojie Zhang, and Dr. Jooheung Lee , for spending their time to review the manuscript and providing valuable comments.

I would like to thank my group members: Pengju Shang, Christopher Mitchell, Grant Mackey, Saba Sehrish and Huijun Zhu . I want especially to thank Pengju, for the inspiring discussions and sharing each step of graduate study with me. I would also like to give a special thanks to Chris and Grant, for their tremendous help during my experiments steps. My gratitude also goes to Huijun Zhu, who's previous work provides great inspirations of this thesis. She also proof-read my paper and provided many suggestions to help me improve it.

I dedicate this thesis to my family: my parents Yongxin Zhang and Yanfu Ma, my girfriend Lisa Li, for all their love and encouragement through my life. I would also like to extend my thanks to my friends, who have cared and helped me, in one way or another. My graduate studies would not have been the same without them.

Finally, I would like to thank the NSF for sponsoring this work under grants CCF-0811413 and CAREER CCF-0953946. Additionally, I thank the US Department of Energy for sponsoring us under Early Career Principal Investigator Award: DE-FG02-07ER25747.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation

The super data cluster has emerged as an ever-popular infrastructure in today's computational architecture research. Recent years have witnessed a growing demand for large-scale data centric clusters. Mountains of data are generated every single day by both scientific and commercial applications. In astronomy, Sloan Digital Sky has stored data for 215 million unique objects and they are still growing [slo]; in geophysics, billions of photos are captured of earth's surface and it is still growing [ear]. At the same time, corporations like Yahoo [yah], Google [goo] and Facebook [fac] have set up their data cluster centers to maintain the massive amount of internet based data. As a case, Facebook [fac] is generating 55,000 images per second at peak usage [Vaj09], all of which require storage of the data. Under these circumstances, researchers and designers are developing ever-larger storage systems to meet the exploding demand of data storage. In systems like these, reliability of the data becomes a great concern because when a super data cluster contains thousands or tens of thousands of nodes, node failure will be a daily occurrence instead of a rare situation [XSM05]. For some systems, such as email servers and etc., minor data loss may be affordable. However, for certain large storage systems, such as scientific-based storage systems, data loss is intolerable, and would result in serious consequences. For instance, if one node containing only a part of a large file failed, the whole file would be unavailable. Thus resulting in a failure effecting the other nodes which store the other sections of the file.

To attack this problem, more and more clusters are turning to multi-way replication based storage architectures due to the high availability requirement. Fortunately, recently the capacity of a single magnetic disk has reached TB levels [1TB07], so storage efficiency is becoming less of a consideration item for building super data clusters than before. Hence, multi-way replication is a more attractive candidate than parity for redundant storage due to its simplicity and effectiveness. For example, Google File System (GFS) [S 03] and Hadoop File System (HDFS) [Bor07] is adopting three-way replication.

However, in replication based architectures, a general but important question is yet left open—before recovery of the failed node, how to find the whole content of the failed node. The answer to this problem is the key in recovering the data immediately. We call this specific process as ***reverse lookup (RL)***. We define this by considering it to be a converse to data distribution, which locates the storage node for one piece of data. The RL process aims to locate all the data based on a failed node ID. The current solutions of RL problem can be divided into two categories—metadata traversing and data distribution reversing. For simplicity, we use the term "object" to symbolize the data unit such as "block" or "chunk" in different systems and the term "node" to symbolize the "storage node".

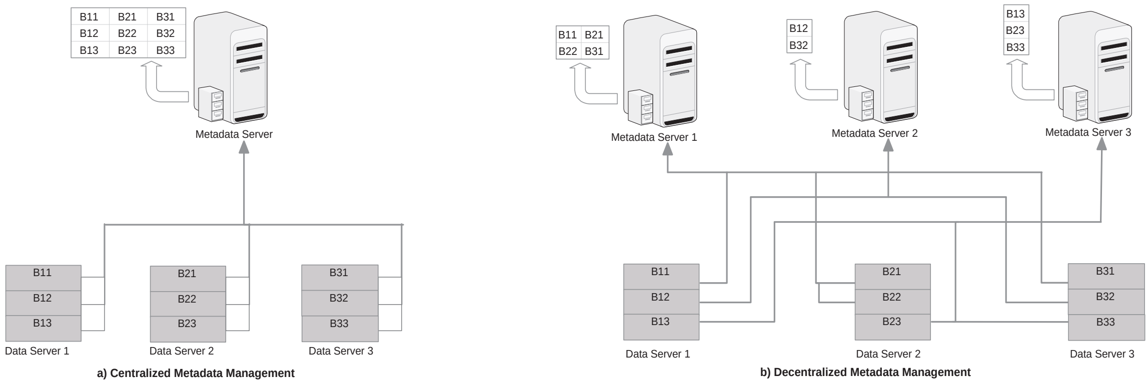


Figure 1.1: Metadata Management Scheme

- *Metadata Traversing*, adopted by systems such as [BO03, CF96], is finding the object list by traversing the metadata index node (inode) table. Metadata in a file system includes dentries and inodes. Dentries maintain the metadata namespace and inodes maintain the file's information such as attributes and block locations. An inode is information about one file or block, while an dentry is the information of the relationship with other files and directories. As the relative information,"node ID" and "object ID", are all held within the inodes, the RL process can retrieve the object list by querying the Metadata Server (MDS). The fundamental problem with the approach is that finding the list of one node requires traversing the whole inode table, which is expensive and time consuming. As shown in Figure 1.1 a). Moreover, recently various decentralized metadata management schemes are becoming employed in an effort to solve the central bottleneck issues of concurrent access. In this architecture, metadata of different objects are mapped to different MDSs. The RL process would become extremely inefficient because one node failure would result in scanning through all MDSs to construct its object list. As shown in Figure 1.1 b).

- *Scalable Data Distribution Reversing* Since RL is the exact reverse process of data distribution, a more efficient solution is to locate the content by reversing the data distribution algorithms. This approach has a major advantage in that, the inefficient and time consuming traversing process can be avoided. However, existing scalable data distribution algorithms include table based (hierarchical lookup table) approaches [SSD96] and computation based (pure hashing) approaches [CF96, WBM06a]. Table based approach supports simple and efficient reverse data lookup. The major limitation using table based RL is the central control and limited scalability. Pure hashing has scalability advantages but there are some drawbacks in that the hashing functions are typically irreversible.

- *Deterministic Data Layout Reversing* Deterministic data layout is developed to place replicas into different disks to achieve high reliability in a replication based architecture. Representative examples are including chained declustering [HD90], group-rotational declustering [CT96], and shifted declustering [ZGW08]. In these schemes, RL is supportive because the objects placement algorithms are reversible. However, these data layouts are designed for small-scale, centralized metadata management architectures—the object's locations are depending on the total number of disks, when the system expands, adding storage nodes into an already balanced layout would result in a large amount of data reorganization. As proved in paper˜citeshifteddeclustering, SD layout is have more data reorganization

1.2 Contribution

In this thesis, we propose a ***Group-based Shifted Declustering (G-SD)*** layout scheme, a scalable placement-ideal layout that leverages our shifted declustering (SD) data layout [ZGW08], and realize scalable and fast RL processes. Shifted declustering data layout is a placement-ideal layout to maximize data recovery parallelism in small-scale, centralized storage architectures. In the SD data layout, object's locations are calculated in specific disks and offsets, hence RL can be achieved by conversing the computation. However, this scheme is designed for small-scale and is not applicable to the large scale file system due to the scalability issue. As stated above, the object's locations are depending on the total number of disks, large data reorganization will happen when disk number varies. G-SD is proposed to solve this particular problem by capping the reorganization overhead into a reasonable level such that this layout scheme can be extended into the large scale file system. This thesis makes the following contributions:

1. We present a scalable Group-based shifted declustering layout that extends the traditional SD so that it can be applied in the large scale file system. It grouped the SD layout and limited the reorganization overhead into a reasonable level.

2. We present a simple but efficient storage node addition algorithm to further reduce the reorganization overhead by introducing "lazy update policy" into the G-SD design.

3. We provide the optimal group size by attacking the tradeoffs between the recovery parallelism and reorganization overhead.

4. We mathematically prove that G-SD is scalable and can be applied in a large-scale file system by computing the overhead G-SD involves for node addition and deletion. From the proof, G-SD incurs a low and reasonable reorganization overhead for node addition and deletion.

5. We examine the proposed G-SD reverse lookup by real-life experiments and prove that G-SD RL outperforms existing solutions including centralized and decentralized metadata traversing methods. Based on our experiment results, the response speed of G-SD RL is up to one order of magnitude faster and on average four times faster than the response speed of centralized metadata traversing. Comparing with decentralized metadata traversing, G-SD response speed is also up to one order of magnitude faster and on average three times faster.

1.3 Organization of the Thesis

The thesis is organized as follows: In Chapter 2, we discuss the background of metadata management methods which are related to the RL process and the shifted declustering layout scheme.

In Chapter 3 we describe the design Group based Shifted declustering layout in detail. In Section 3.6, we present the reverse lookup process of G-SD.

Chapter 4 analyzes the reorganization overhead and the performance evaluation for G-SD.

Finally, we make the conclusion remarks in Section 5.

CHAPTER 2

BACKGROUND AND RELATED WORKS

Based on our investigation and to the best of our knowledge, the reverse lookup problem has not been widely studied. Partially because in some cases, this problem is not vital, and there is no need to recover or reconstruct the node when node failure happens. For example, Peer-to-Peer systems, such as Chord [SML03], are assumed insecure and employed high degree of replication. Most of the systems make little attempt to guarantee long persistence of stored objects. In these systems, nodes are added and removed frequently. Losing one node simply means to lose access to the data store on that node while other nodes are maintaining the same data and would be able to provide query services. Kademlia [MM02] keeps k replicas,which is usually set to 20, for one file. Under these circumstances, recovery and reverse lookup is unnecessary.

Some other systems such as GFS [S 03], adopt a “lazy recovery” scheme—it maintains a certain threshold for chunk replicas. When the number of replicas for certain blocks, are lower than the predetermined threshold which may be resulted from node failures, the system will initiate a replication for the missing chunk(s). In this situation, reverse lookup is not needed because GFS [S 03] is preserving a “chunk level” recovery instead of the “node level” recovery. Indeed, this “lazy recovery” scheme could reduce the importance of the reverse lookup problem, however, maintaining a threshold for each object is still generating a large overhead, which using G-SD can avoid.

The reveres lookup problem resides in the system that follows the basic assumptions below:

1. The file systems is at least a petabyte-scale storage system, which contains thousands or tens of thousands of storage nodes. Each node holds a large number of data objects. Objects are relatively large in terms of tens of Megabyte to several Gigabytes.

2. The storage servers and clients are tightly connected: this is different from the loosely connected architecture such as peer-to-peer networks which communication latency varies from node-to-node.

3. The system requires a “node-level” recovery instead of the “object level” recovery.

4. A declustered k-way replication scheme is employed to keep up the availability and reliability.

As mentioned in Chapter 1, there are two basic solutions for this problem: metadata traversing and data distribution reversing.

Table 2.1: Comparison between Existing Solutions in finding object list for node i, where N and M are total number of metadata entries and metadata servers, repectively. $O(p)$ is the time for piecing together list results. B and N' are total number of blocks of node i and number of nodes within a G-SD group, respectively.

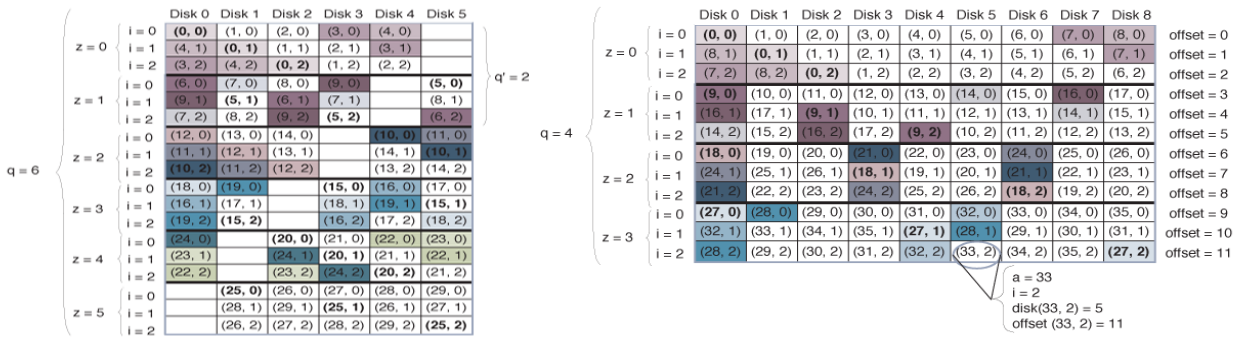
		Examples	Feasibility	Parallelism	RL Time	Storage Overhead
Metadata Lookup	Centralized	Lustre	Yes	No	$O(N)$	$O(i)$
	Decentralized	PVFS2	Yes	Yes	$O(\frac{N}{M}) + O(p)$	$O(i)$
Scalable data distribution reversing	Central directory	xFS	Yes	No	$O(logN')$	$O(i)$
	Tree-based	DRT, RP*	No	N/A	N/A	N/A
	Hash-based	LH*, RUSH, CRUSH	No	N/A	N/A	N/A
G-SD RL			Yes	Yes	$O(\frac{B}{N'})$	0

2.1 Metadata Traversing

Metadata traversing Metadata traversing approaches vary among different metadata management schemes. Existing metadata management schemes could be divided into two categories: centralized and decentralized metadata management. In centralized metadata management systems, such as Lustre [BO03], the whole namespace in these two systems are in one or duplicated on multiple metadata servers. Reverse lookup process is able to be conducted by traversing the whole inode table—suppose the system wants to construct the block list i, the operation is "find the inode which has the owner ID i". As shown in Figure 1.1 (a).

Figure 1.1 (b) shows the decentralized metadata management scheme. In this scheme, the system distributes the file metadata into multiple MDSs using different strategies, such as table-based mapping, hash-based mapping, static-tree based mapping, dynamic-tree based mapping and bloom filter-based mapping. As file metadata from the same node could be distributed into different MDSs, the decentralized metadata management scheme divided the system namespace into different parts. In this way, metadata queries and workloads are shared so that central bottleneck can be effectively avoided. In the case of the above example, to retrieve the object list for node i, all metadata from every MDSs have to be examined and the result from each MDS has to be pieced together.

However, for both architectures, this traversing process significantly increases the recovery time because examining such a large amount of metadata for a petabyte-scale system will take a long time. Even though parallel discovery in decentralized metadata management systems may be introduced to reduce the lookup time, piecing the on-disk data structures back together will take considerable time. To make things worse, this time consuming process also increases the probability of data loss due to the fact that other nodes may fail during the time of traversing and examining metadata. As stated in paper [SSD96], "file systems of petabyte-scale and complexity cannot be practically recovered by a process, which examines the file system metadata to reconstruct the file system. " This long lookup time makes this scheme not suitable for fast recovery.



a) Example layout for k=3,n'=6

b) Example layout for k=3,n'=9

Figure 2.1: Example for Shifted Declustering Layout

2.2 Scalable Data Distribution Reversing

Scalable Data Distribution Reversing A more reasonable solution for reverse lookup is to reverse the process of data distribution algorithms, which aims to provide the proper location for a piece of data. i.e. RUSH [HM04] is a decentralized data distribution algorithm that could maps replicated objects to a scalable collection of storage servers or disks. Data distribution approaches could also be divided into two categories: centralized control(table based) and decentralized control.

The centralized control approach, adopted by file systems such as xFS [WWA93], also known as table based mapping method, uses a centralized directory for locating data in different storage nodes. Reverse lookup could be easily implemented by scanning and querying the location directory due to the fact that tables and directories are reversible. However, this method itself is not suit for a petabyte-scale file system—in a large-scale cluster built from thousand of individual nodes, the distributing data based on a centralized directory would result in a severe bottleneck which would dramatically reduce the overall performance.

Decentralized control approaches, also called scalable distributed data algorithms, have come about in recent years to solve the bottleneck problem in centralized architecture. The algorithms in the decentralized control approaches distribute data following three properties which allows them to provide good performance in large, scalable environment—dynamic, decentralized and no large update.

1. The algorithm should be *dynamic*, which means the number of servers sharing the file should adapt gracefully to the number of keys to be stored.

2. The algorithm should be *decentralized*, meaning that no master site should be involved. e.g. centralized directory mentioned above.

3. The algorithm should not incur large update overhead. This means that at any point in time, not too many nodes should be forced to handle messages that inform about changes, and to adapt themselves to these changes.*i.e.* optimal location reorganization.

While scalable data distribution algorithms are mainly utilizing two techniques: trees and hashing, both methods have problems in adopting the reverse lookup problem. On one hand, tree based distribution algorithms, including Distributed Random Trees(DRT) [KLL97] and RP* [LNS94], are not supporting data replication which is a must for our recovery scheme. On the other hand, though hash based distribution algorithms, such as LH* [LR02], RUSH [HM04], CRUSH [WBM06b] and etc., support data replication, the hash function itself is irreversible—given a node ID, we could not retrieve the block list that has been distributed based on the distribution algorithms.

As summarized in Table 2.1, all the existing methods are either too time consuming or not supportive enough for the reverse lookup problem. Our previous work SD layout can achieve fast reverse lookup problem but it is not suitable for applying into the large-scale system. In this thesis, we propose Group-based Shifted Declustering (G-SD) data layout, to efficiently place objects and replications and solve the flexibility and scalability of SD. It obeys all three properties meanwhile support the reverse lookup process within a relatively small overhead.

CHAPTER 3

GROUP-BASED SHIFTED DECLUSTERING DESIGN

In this section, we present a novel approach, called Group-based Shifted Declustering (G-SD), which is an extension of Shifted declustering (SD) data layout scheme and can carry out a recovery-oriented optimal data placement and can support the efficient reverse lookup process. SD layout, shown in Figure 2.1 is a recovery-oriented placement layout scheme in multi-way replication based storage architectures, which aims to provide better workload balancing performance and achieve maximum recovery performance when node failure occurs. As illustrated in [ZGW08], SD satisfied all six properties of placement-ideal layout and can reach at most (n-1) parallelism in recovery. By using this scheme, the recovery time can be reduced and thus shorten the “time of vulnerability”. However, applying this scheme in large-scale file systems would require one more functionality—optimal reorganization. The SD algorithm is dependent on the total number of disks that exist in the system. When new storage nodes are added into the system, most objects need to be moved in order to bring a system back into balance and optimal. This complete reshuffling process, which may take a system offline for hours or even days, is unacceptable for our systems [HM04]. For instance, a 1 petabyte file systems built from 2000 disks, each with a 500 GB capacity and peak transfer rate of 25 MB/sec would require nearly 12 hours to shuffle all of the data. Moreover, the reorganization process would require an aggregate network bandwidth of 50GB/sec. In contrast, a system running G-SD can complete reorganization in a rather less amount of time because only a small fraction of existing disk bandwidth is needed to participate in the reorganization.

3.1 Definitions and Notations

Before the design, we define several terminologies, as follows.

1. The *redundancy group* is the set of a object and all its replicas.
2. The *Stable group (sub-cluster)* is a group that has stable number of nodes and will not influenced by the system expansion. The number of nodes in a stable group is fixed after generated.
3. The *unstable group (sub-cluster)* is a group that does not have a stable number of nodes and will be influenced by the system expansion. The number of nodes in an unstable group will change when adding new nodes into the system.
4. The *group reorganization* is the process of moving objects to other storage nodes in order to keep the original layout after new storage nodes are added into the group.

The notations are summarized in Table 3.1. There are four system configuration parameters for the G-SD layout: the total number of storage nodes in the system m ($m \geq 2$), number of storage nodes in a stable sub-cluster n ($n \leq m$), number of storage nodes in a unstable sub-cluster n' ($n' \leq n$) and the number of units per redundancy group k ($k \leq m$). Within a redundancy group, the units are named from 0 to $k-1$, and we view the unit 0 as the object (primary copy), and other units as replica units (secondary copies). Distinguishing objects from their replica is only for the ease of representation, although they are identical. We use an address a ($a \geq 0$) to denote a redundancy group, and a can also be considered as a redundancy group ID. Each unit is identified by the name (a, i), where a is the redundancy group ID, and $0 \leq i \leq k-1$. Without the loss of generality, $(a, 0)$ represents the object, and (a, i) with $i > 0$ represents the i-th replica unit. The location of the unit (a, i) is represented by a tuple $(\mathbf{disk}(a, i), \mathbf{offset}(a, i))$. A complete round of layout is obtained by q iterations, and in each iteration, one row of data units and $k-1$ rows of replica units are placed, so that the total rows of units in a complete round is $r = kq$. Repeating complete rounds of layouts also yields placement-ideal layouts. In the following, we only consider the layouts within a complete round.

3.2 Group-based Shifted declustering Scheme

The G-SD is proposed to minimize the reorganization overhead of shifted declustering layout so that this layout can be utilized in large-scale file systems. It is based upon a simple principle: storage nodes or disks are divided into sub-clusters so that when nodes are being

Table 3.1: Notation summary

Symbols	Descriptions
m	Total number of data node in the cluster
n	Number of stable data node in each sub-cluster
n'	Number of Unstable data node in each sub-cluster
k	Number of units per redundancy group
a	The address to denote a redundancy group
(a, i)	The i-th unit in redundancy group a
q	Number of iterations of a complete round of layout
y, z	Intermediate auxiliary parameters
$\mathbf{disk}(a, i)$	The disk where the unit (a, i) is distributed
$\mathbf{offset}(a, i)$	The offset within $disk(a, i)$ where the unit (a, i) is distributed

added or removed, reorganization process would only influence the nodes within one group rather than the whole file system, as traditional shifted declustering does.

In G-SD, suppose we have m nodes in the system and we divide them in to multiple stable groups with each group containing n nodes and one unstable group with n' ($n' \leq n + k + 1$) nodes. In each group, a shifted declustering scheme is applied to optimize the replica placement so that maximum recovery performance can be achieved when node failure occurs. That is, if node j in group a failed, the number of nodes that could participate in the recovery is maximized within the group. While members in one group are correlated

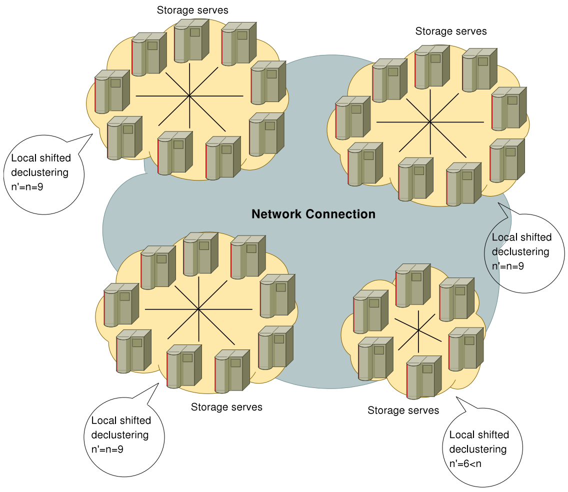


Figure 3.1: Group-SD Scheme

with each other, groups are independent— each holding disjoint fractions of objects for the whole system. Replicas will not be placed among different group members. By keeping little relationships between different groups, reorganization process brought by adding new nodes into the system will only influence the members within one group, leaving other groups unaffected. Figure 3.1 shows an example of G-SD scheme: nodes are divided into different sub-clusters. Sub-cluster 1, 2, and 3 are all consists of 9 nodes, which is the stable group number, while sub-cluster 4 is unstable group with 6 nodes. Each sub-cluster is managed by local shifted declustering layout—1, 2, 3 are shown as Figure 2.1 a), 4 is shown as Figure 2.1 b), respectively.

3.3 Efficient Reorganization for Group Addition

While grouping can reduce the time consumption and reorganization overhead for large-scale file systems, it needs improvement based on the observation mentioned in paper [HM04]—when the large storage systems is expanded, and new capacity is typically added multiple disks at a time, rather than by adding individual disks. If updating occurs immediately after one new node comes, extra reorganization overhead will be introduced. In this case, we introduce the "Lazy policy". That is, the reorganization process will wait until all newly joint nodes are fully integrated into the system to decide which group they join.

Figure 3.2 illustrates the flow of adding new nodes into the system. When one node λ_1 joins the system, it does not directly choose any sub-cluster. Instead, the cluster maintains a tracking list for newly added nodes. Then if new disks $\lambda_2, ...\lambda_l$ are added into the cluster, the operation conducted is the same as λ_1. After the new nodes are fully integrated, finished adding, groups are determined by the number of nodes added: 1) we prioritize the group-based addition. If $l > n$ (l is the newly added nodes, n is the stable group size), $\lfloor l/n \rfloor$ new stable groups are added into the system with each group having n nodes. 2) After the stable group addition, if the remaining $l\%n + n'$ (n' is the unstable group size) newly added nodes is no less than the stable group size n ($l\%n + n' \geq n$), then a stable group will be formed with n' old nodes and $n - n'$ new nodes, remaining $l\%n - n + n'$ nodes will be formed as an unstable group. If $l\%n + n' < n$, the $l\%n$ newly added nodes will join the unstable group.

For instance, in a three-way replication system the optimal sub-cluster size n (Section 3.5) is 10 and there are 27 nodes already in the system, which are subsequently divided into three groups (2*10 nodes stable groups, 7 nodes unstable group). If adding 23 new nodes into the cluster, the newly added group will be divided into two stable groups first, which contains 10 nodes each. The remaining 3 nodes will join the existing unstable group to produce a 10 nodes stable group. If 27 nodes are added into the cluster, the remaining 7 nodes will be divided into two parts, first, three nodes will participate in the former unstable group to form a 10 nodes stable group. The left 4 nodes will form a unstable group due to the fact that there is a requirement for three nodes as the minimum number $n \geq 4$ for each sub-cluster to apply the local shifted declustering layout. Since node addition only maintains the information about the number of newly added nodes and this process takes little time, the total storage requirement of tracking this list could be negligible. By using the "lazy policy", tracking list and group "prior" methods, we are able to reduce the reorganization overhead by merging several reorganization process into one or even totally eliminating the reorganization process in some cases (will mention in Section 3.4).

We consider disk retiring the same as disk recovery because the retiring disk has to "shed" its data to other data nodes which will results in a temporary local shifted declustering unbalance. Though disk departure could be supported by our solution, it cost extra overhead to reorganize the replicas in an already balanced shifted declustering layout sub-cluster. Since in our solution, recovery is parallelized and efficiency, we believe that substituting the retired node and pour all the data to the new node is a better idea than nodes departure and

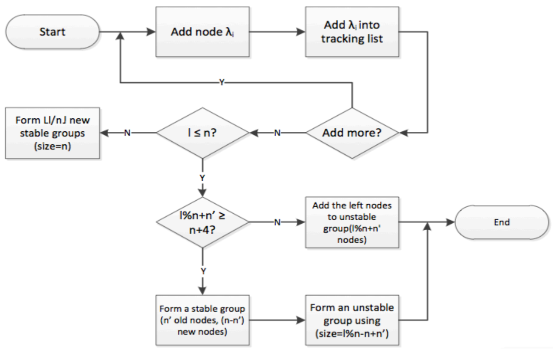


Figure 3.2: The Procedure of “Lazy Node Addition Policy”

rebalancing of loads. Before substitution, if some nodes are not available, the workload could still be evenly distributed into the existing nodes, which will not degrade the performance in a great manner.

3.4 Group Addition Guideline

By using the above schemes, we considerably reduce the reorganization overhead. In this section, we give a simply and practical recommended configuration that allows the cluster to totally eliminate the reorganization costs without sacrificing the advantage of local shifted declustering. We recommend to add or removing of groupings in times of n nodes at a time. Since each sub-cluster is not related to others in the manner of object placement, adding or removing exactly a group will not trigger the reorganization process. All newly added nodes are formed as new stable groups.

G-SD does not recommend "Group merging or Splitting" [HZJ10] to add or remove new nodes due to two reasons. One main reason is that these approaches involve too much overhead and can not efficiently support the group reconfiguration. Group merging is the merge of two groups when their sizes are both lower than $\lfloor n/2 \rfloor$. Group splitting process is equivalent to deleting $\lfloor n/2 \rfloor$ nodes DS from the existing sub-cluster and inserting them into new groups. Applying deletion in an already balanced shifted declustering group will result in reorganization. Also, it requires much more overhead to optimize two groups $\lfloor n/2 \rfloor$ and $n - \lfloor n/2 \rfloor$ than a newly generated group with much fewer nodes in it.

3.5 Optimal Group Size Configuration

Since the number of a stable group will not change after it is formed. It is vital and one of our key design issues in G-SD is to identify the optimal stable group number n. n can strike different tradeoffs between node failure rate and recovery parallelism. As n increases, recovery parallelism within one group is increasing and thus lower recovery and degradation time. In an extreme case, when the nodes are all in one group, $n = m$, recovery parallelism can reach to the maximum $m-1$. A larger n, however, typically leads to a higher possibility of disk failures. For instance, if the failure rate for one node is r, the failure rate for n nodes is increasing to $n * r$. In this section, we discuss how to obtain the optimal n for a stable group.

To identify the optimal n, we use a simple benefit function that jointly present failure rate and recovery parallelism using overall recovery time. Since recovery process will both degrade the system normal performance, we aim to determine an optimal stable group size n that can minimize the total performance degradation time μ and as a result the sub-cluster normal performance time is maximized. Equation 3.1 shows the function to evaluate the sub-cluster's availability.

$$\mu = P_{Fail} * T_{Recovery} \tag{3.1}$$

where P_{Fail} is the probability of node failure happens in the whole system. $T_{Recovery}$ is the time for recovering the failed node.

The probability of node failure is associated with group size n and failure rate per disk r, which is shown in Equation 3.2.

$$P_{Fail} = C_n^f * r^f * (1-r)^{n-f}, f \in [1, \lfloor \frac{n}{2} \rfloor] \tag{3.2}$$

where f is the number of failed nodes in the same time. The maximum number of failed nodes f is $\lfloor \frac{n}{2} \rfloor$ (n is the total number of nodes in the stable group) otherwise the whole group will fail and can not be recovered.

The recovery time is influenced by the recovery parallelism. As objects is evenly distributed in one group using local shifted declustering layout. All other nodes in the same group will participate in the recovery in a parallel manner. Assume t is the time for recovering one failed node sequentially, the parallel recovery time is represented in Equation 3.3.

$$T_{Recovery} = \frac{tf}{n-f}, 1 \le f \le \lfloor \frac{n}{2} \rfloor \tag{3.3}$$

So the optimal value of stable group size n is the one that could minimize the μ in Equation 3.1, as shown in Equation 3.5.

$$\begin{aligned}
\mu &= P_{fail} * T_{recovery} \qquad (3.4)\\
&= \sum_{f=1}^{\lfloor \frac{n}{2} \rfloor} C_n^f * r^f * (1-r)^{n-f} * \frac{tf}{n-f}\\
&= C_n^1 * r * (1-r)^{n-1} * \frac{t}{n-1} + C_n^2 * r^2 * (1-r)^{n-2} * \frac{2t}{n-2} + C_n^3 * r^3 * 1 - r^{n-3} * \frac{3t}{n-3}\\
&\quad +... + C_n^{\lfloor \frac{n}{2} \rfloor} * r^{\lfloor \frac{n}{2} \rfloor} * (1-r)^{n-\lfloor \frac{n}{2} \rfloor} * \frac{\lfloor \frac{n}{2} \rfloor t}{n - \lfloor \frac{n}{2} \rfloor}
\end{aligned}$$

where we ignore the cases that disks fail in the middle of a recovery process.

As t and r are all constant, to minimize μ we could calculate the derivative of Equation 3.5, as shown follows.

$$\begin{aligned}
\mu' &= \left(\sum_{f=1}^{\lfloor \frac{n}{2} \rfloor} C_n^f * r^f * (1-r)^{n-f} * \frac{tf}{n-f} \right)' \qquad (3.5)\\
&= tr * \left(\frac{(1-r)^{n-1} n}{n-1} \right)' + \frac{2tr^2}{2!} * \left(\frac{(1-r)^{n-2} n(n-1)}{n-2} \right)'\\
&\quad + \frac{3tr^3}{3!} * \left(\frac{(1-r)^{n-3} n(n-1)(n-2)}{n-3} \right)' + \frac{4tr^4}{4!} * \left(\frac{(1-r)^{n-4} n(n-1)(n-2)(n-3)}{n-4} \right)' +\\
&\quad ... + t * \left(\frac{\lfloor \frac{n}{2} \rfloor r^{\lfloor \frac{n}{2} \rfloor} (1-r)^{n-\lfloor \frac{n}{2} \rfloor} n(n-1)(n-2)...(n-\lfloor \frac{n}{2} \rfloor + 1)}{\lfloor \frac{n}{2} \rfloor ! (n - \lfloor \frac{n}{2} \rfloor)} \right)'
\end{aligned}$$

where we call $u' = 0$ to retrieve the extrema, and get n from the equation with one unknown variable.

For example, we choose $r = 3\%$ per year. According to Schroeder *et al.*'s observation [SG07], in a real-world large storage and computing systems, the ARR (annual replace rate) of hard drives is between 0.5% and 13.8%, and 3% on average. The reasonable and optimal n computed from Equation 3.6 is 9 for n is odd and 8 for n is even.

3.6 G-SD Reverse Lookup

As mentioned in Chapter 1, disk failure becomes the norm rather than exception. Storage system is designed to work uninterruptedly under some or even the entire failed disks, only if there is enough redundancy to reconstruct. However, to keep the long-term functionality, it is desired to recover the failed components as soon as possible (MTTR), *i.e.* active repair. To attack this problem, one important issue is to understand "what need to be recovered" and know it in a quick way. Thus fast reverse data lookup, which is the process locating a specific object in a given location, should be conducted.

When a node fails in a cluster, reverse lookup in G-SD may involve two hierarchical levels: acquiring the group ID in which the failed node exists, and retrieve the object list from the local shifted declustering reverse function. This hierarchical design is to provide fast and guaranteed reverse lookup that will find out the objects that needs to be recovered in a fairly short time.

Each query is performed in the following sequences. First, one node failure is detected by heartbeats, it will return the tuple $< groupID, nodeID >$. Obtaining the tuple containing the group ID, system could conduct the reverse lookup within a specific group. Second, retrieve the object list by using the shifted declustering inverse function. We would like to give an example for three-way replication. Given the notation in Table 3.1, the reverse lookup algorithm can be abstracted as: assuming k (the number of units per redundancy group) and m' (the number of disks in the sub-cluster) are known system parameters, d

(disk/node ID) and o (offset) are given location, find out the redundancy group ID a located on disk d and offset o.

In three-way replication with $n = 4$ or n is odd, the local reverse lookup function could be summarized in through Equation 3.6 to Equation 3.10.

$$q = \begin{cases} 1, \text{ if } n = 4 \\ (n-1)/2, \text{ if } n' \text{ is odd} \end{cases} \tag{3.6}$$

$$i = o\%k \tag{3.7}$$

$$z = \lfloor a/n \rfloor \tag{3.8}$$

$$y = (z\%q) + 1 \tag{3.9}$$

$$a = \begin{cases} d - (zn + iy)\%n + zn, \text{ if } (zn + iy)\%n \leq d \\ d - (zn' + iy)\%n + (z+1)n, \text{otherwise} \end{cases} \tag{3.10}$$

where Equation 3.6 to 3.9 is intermediate variables, and Equation 3.10 computes the redundancy group ID a.

In three-way replication with $n > 4$ and n is even, the algorithm could be formulated as:

$$n' = n - 1 \tag{3.11}$$

$$q' = (n' - 1)/2 \tag{3.12}$$

$$i = o\%k \tag{3.13}$$

$$z = \lfloor a/n' \rfloor \tag{3.14}$$

$$d_b = (n' - z)\%n' \tag{3.15}$$

$$y = (z\%q') + 1 \tag{3.16}$$

if ($\mathbf{disk}(a, 0) < d_b$ and $\mathbf{disk}(a, 0) + iy \geq d_b$)

or $\mathbf{disk}(a, 0) > d_b$ and $\mathbf{disk}(a, 0) + iy \geq d_b + n'$)

if $(zn + iy + 1)\%n \leq d)$

$a = d - (zn + iy + 1)\%n' + zn$

else $a = d - (zn + iy + 1)\%n + zn$

else if $((zn + iy)\%n \leq d)$

$a = d - (zn + iy)\%n + zn$

else $a = d - (zn + iy + 1)\%n + (z + 1)n$

In the above equations, a is different due to the fact that the placement for $\mathbf{disk}(a, i)$ varies before or after the bubble that mentioned in [ZGW08].

CHAPTER 4

ANALYSIS AND EXPERIMENT RESULTS

4.1 Reorganization Overhead

G-SD is aiming to take the advantage of fast and efficient reverse lookup of the shifted declustering algorithm and minimize its reorganization overhead. Here we explicitly give the overhead of adding new nodes into a shifted declustering layout and compare it with system using G-SD. For a three-way shifted declustering layout, assume there are originally α nodes and we are about to add β nodes into this layout. The total number of objects that needs to be reorganized is as follows.

$$R(\alpha,\beta) = \begin{cases} [(q-1)\alpha + (\beta+1)] \times 3,^{*} \\ [(q-1)\alpha + (\beta+1)] \times 3 + 2,^{**} \\ [(q-1)\alpha] \times 3,^{***} \\ [(q-1)\alpha + \beta] \times 3,^{****} \\ [(q-1)\alpha + \beta] \times 3 + 2,^{*****} \end{cases} \tag{4.1}$$

*: if α is odd and β is even

**: if α is even and β is odd

***: if α is odd and $\beta = 1$

****: if $\alpha > 1$, α is odd and β is odd,

*****: if α is even and β is even,

In Equation 4.1, $q = 1$ if α is 4, $q = (\alpha - 1)/2$ if α is odd, or $q = (\alpha - 2)/2$ if v is even. $R(\alpha, \beta)$ is mainly determined by the number of objects that needs to be moved from one node to another. The proof is provided in Appendix A. Suppose the overhead of moving one object to another node is v, the total reorganization overhead is $v * R(\alpha, \beta)$ without considering the parallelism. However, the overhead is usually measured by time. Based on the observation in [VDF02], the data reorganization process is made in sequential for nodes filled with data. For the newly added nodes, the reorganization process can run in parallel. Assume the number of newly added nodes will not exceed the number of current nodes. The number of sequentially moved objects is shown in Equation 4.2.

$$R_s(\alpha, \beta) = \lfloor \frac{3\alpha * (q-1)}{\alpha + \beta} * \alpha \rfloor \tag{4.2}$$

The number of parallel moved objects is shown in Equation 4.3.

$$R_p(\alpha, \beta) = \lfloor \frac{3\alpha * (q-1)}{\alpha + \beta} * \beta \rfloor + 3\beta \tag{4.3}$$

Suppose the time of moving one object to another is e, based on the above equations, we compare the reorganization overhead and time between SD and G-SD layout.

1. In a systems with SD layout, the total number of nodes is m. When l nodes are added into the system, the total overhead of balancing the shifted declustering layout is $O_{SD} = v * R(m,l)$. The reorganization time is $T_{SD} = e * [R_s(m,l) + R_p(m,l)/l]$.

2. In G-SD with total m nodes, at most one unstable group is effected when adding nodes into the system. Thus the reorganization overhead will be limited to the unstable group size $n' < n+4$. Specifically, when adding l new nodes into this system, the total overhead is shown as follows.

$$O_{GSD} = \begin{cases} v * R(n', n-n'), \text{if } l\%n + n' \geq n+4 \\ v * R(n', l\%n), \text{if } l\%n + n' < n+4 \end{cases} \tag{4.4}$$

   As shown in Figure 3.2, when $l\%n+n' \geq n+4$, the reorganization overhead is generated from forming one stable group using n'original unstable group nodes and $(n-n')$ newly added nodes. When $l\%n+n' < n+4$, the overhead is generated from adding $l\%n$ new nodes into the original unstable group (n' nodes). The reorganization time is shown in Equation 4.5.

$$T_{GSD} = \begin{cases} e * [R_s(n', n-n') + R_p(n', n-n')/l]\,, \text{if } l\%n + n' \geq n+4 \\ e * [R_s(n', l\%n) + R_p(n', l\%n)/l]\,, \text{if } l\%n + n' < n+4 \end{cases} \tag{4.5}$$

Figure 4.1 and 4.2 present the reorganization overhead and time comparisons between G-SD and SD in different m(system size) and l(number of nodes added). In Figure 4.1, we assume v as 1 (unit). The x axis is the system size m and the y axis is the reorganization overhead. In Figure 4.2, we assume e as 1 *unit*. The x axis is the system size m and the y axis is the reorganization time. In both experiments, we set the number of newly added nodes as 10, 20, 50, 80, or 100.

We can see from Figure 4.1, the reorganization overhead is directly related to the number of nodes in the system m and number of nodes added l. Specifically, in a SD layout with the number of nodes m growing from 4 to 100, every line is maintaining an increasing trend. Also, as the number of nodes added increases, the trend goes sharper. On the other hand, reorganization overhead in a G-SD is in is fluctuating around a small number and not directly effected by m and l.

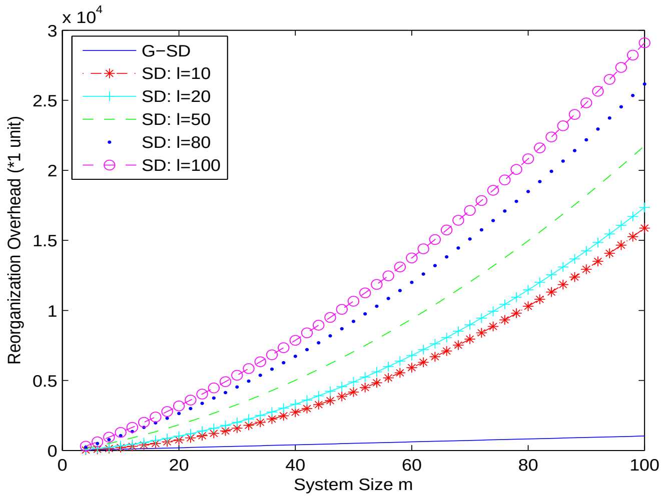


Figure 4.1: Reorganization overhead comparison in different group size (n) and different number of nodes added (l)

As parallelism is introduced into the reorganization process, the time curves for SD in Figure 4.2 are showing big differences from Figure 4.1. Though the time is still increasing with n growing, the time actually decreases with l increasing. This is because when l

increases, more objects will participate in the parallel reorganization and thus reduce the time accordingly.

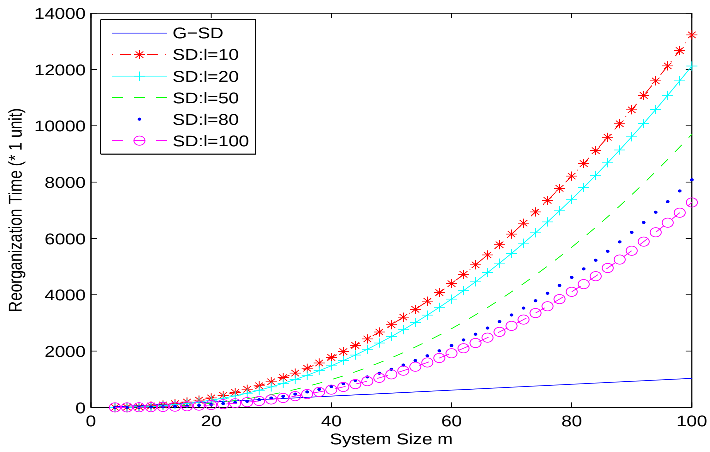


Figure 4.2: Reorganization time comparison in different group size (n) and different number of nodes added (l)

4.2 Evaluating Performance of G-SD Reverse Lookup Schemes

We examine the performance of G-SD through comparing with traversing method in both centralized and decentralized metadata management systems using real-life experiments over a commonly used parallel file system—PVFS2 [pvf]. PVFS2 is an open-sourced parallel file system which is jointly developed by Clemson University, Argonne National Laboratory and Ohio Supercomputer Center. It is flexible to configure different settings of metadata servers (MDSs) and I/O servers. Due to the hardware limitation, we can only set up the experiment environment using CASS cluster which contains 16 nodes, including one head node and

15 compute nodes. Table 4.1 shows the CASS cluster configuration. For example, in our experiments, we configured different number of MDSs and I/O servers. In order to compare between G-SD RL and centralized metadata traversing method, we set up one MDS and 15 I/O servers. When comparing G-SD RL with the decentralized metadata traversing, we set up 10 MDSs and 15 I/O servers.

To evaluate the metadata traversing, we used LLNL's IOR benchmark [lln] to simulate the different data sizes. IOR was developed to set performance targets for LLNL's ASCI Purple system procurement. It measures the system's sequential read/write performance under different file size, I/O transaction size, and concurrency. Moreover, it supports the MPI-IO interface, which can be achieved in PVFS2. In our experiments, IOR is used to generate different sizes of data sets so that we can comprehensively evaluate the metadata traversing time.

After ingesting data into PVFS2, we run a "*time pvfs2-ls -lR*" on the head node to retrieve the metadata traversing time. As mentioned in Section 2, the system has to go through the whole metadata inode entries to retrieve the object list for one storage node. This time is equal to recursively list all the files and objects in PVFS2. Then we get the G-SD RL time by running the RL algorithm and multiplying this time the number of objects in one storage node. Since in PVFS2 the data unit is "block", in the experiments we use "block" instead of "Object" to illustrate the data unit.

Table 4.1: CASS Cluster Configuration

Node Configurations	
Make & Model	Dell PowerEdge 1950
CPU	2 Inter Xeon 5140, Dual Core, 2.23 GHz
RAM	4.0 GB DDR2, PC2-5300, 667 MHz
Internal HD	2 SATA 500 GB (7200RPM) or 2 SAS 147 GB (15K RPM)
Network Connection	Intel Pro/1000 NIC
Operating System	Rocks 5.0 (Cent OS 5.1), Kernel: 2.6.1853.1.14.e15
Cluster Network	
Switch Make & Model	Nortel BayStack 5510-48T Gigabit Switch

4.2.1 Centralized Metadata Traversing *vs.* G-SD Reverse Lookup

In this section, we evaluate the performance through comparing the response latency of proposed G-SD RL scheme with the centralized metadata traversing (CMT) in different configurations, as shown in Figure 4.3. We set the block sizes as 50 MB, 100 MB and 500 MB and the total data size as 5G, 10G, 20G and 50G in IOR benchmark respectively, which automatically ingests the data into PVFS2. Since the traversing process has to go through all the inode entries to retrieve the object list, the response latency of metadata traversing time is mainly influenced by the total number of objects. On the other hand, the response latency of G-SD RL is effected by the number of objects in the failed storage nodes.

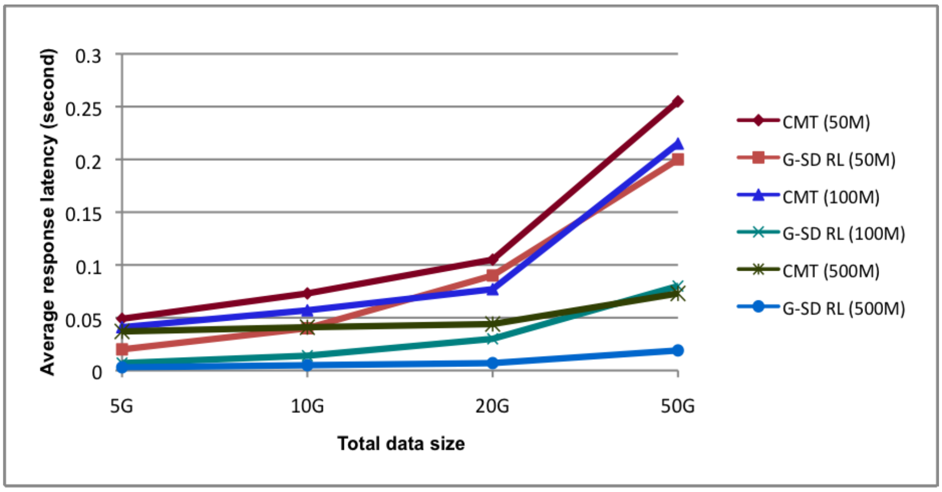


Figure 4.3: Average response latency comparison between G-SD RL and Centralized metadata traversing

We can see from Figure 4.3 that the G-SD RL outperforms the metadata traversing process in every configuration. The curve representing G-SD RL is alway below the curve of CMT, which illustrating that G-SD RL can deliver a faster response speed when node failure occurs. The maximum speedup of G-SD is on order of magnitude faster than CMT when block size is 500 MB and total data size is 5 GB. The average improvement of G-SD is four times faster. The reason for this huge improvement is because G-SD RL is consuming CPU cycles while the CMT is accessing to memories or even disks. Moreover, based on the curve trend, it is reasonable to anticipate that the gap between CMT and G-SD RL is going to increase with the growing data size. For example, when block size is 500 MB, the speedup of G-SD RL increases 100% when the total data size grows from 5 GB to 50 GB. This time difference will be more significant when the system data size reached to petabyte scale.

4.2.2 Decentralized Metadata Traversing *vs.* G-SD Reverse Lookup

Figure 4.4 presents comparison of the average response latency between G-SD RL and decentralized metadata traversing (DMT) in different configurations. In this experiment, we set 10 metadata servers (MDS) and 15 I/O servers in the PVFS2. The other configurations are the same to Section 4.2.1. By running multiple-MDSs, it is anticipated that the traversing time can be reduced because MDSs traversing are working in parallel. However,the other factor, piecing the inode data from different MDSs together, is delaying the response time.

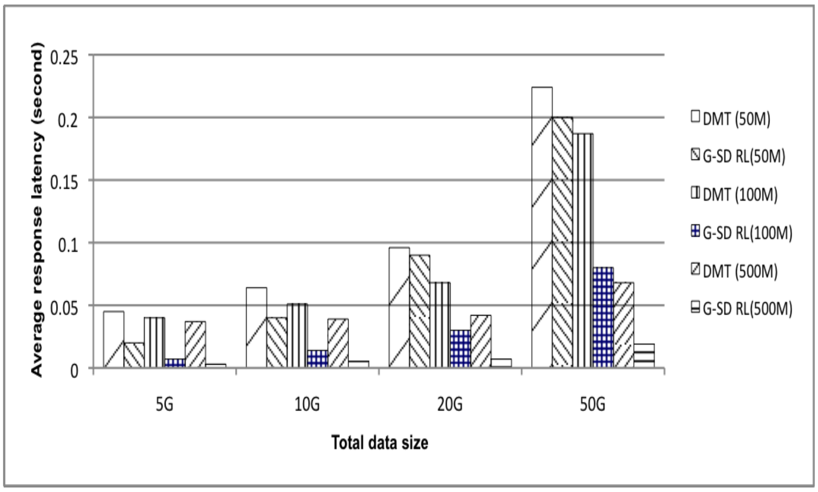


Figure 4.4: Average response latency comparison between G-SD RL and decentralized metadata traversing

Since we can not get the traversing time and piecing time separately, we provide an overall time of the response time. The experiment results show that G-SD RL also shortens the response speed in every configuration. The maximum speedup is also 10 times faster with the configuration of 500 MB block size and 5G total data size. The average speedup is three times faster.

CHAPTER 5

CONCLUSIONS

In this thesis, we have proposed a Group-shifted declustering layout, which can be used in the large file system and support a fast and efficient reverse lookup scheme. We have summarized the importance of the reverse lookup problem, which were overlooked in previous researches. We also examined the current reverse lookup solutions, presented their limitations and proposed a scalable reverse lookup scheme. G-SD extends our previous work shifted declustering so that this multi-way replication-based data layout can be adapted in large file systems. According to the comprehensive mathematical proofs, G-SD is scalable and can largely reduce the data reorganization overhead of SD when adding new nodes. Moreover, by comparing with the traditional reverse lookup method including centralized and decentralized metadata traversing, we conducted real-life experiments to show that the G-SD RL effectively reduced the reverse lookup response time by up to one order of magnitude.

APPENDIX A

Proof of Reorganization Overhead

In Section 4.1, we explicitly present the overhead of adding nodes in an already balanced group. Here we give the proof that how many objects are moved from the original disk to another one. The addition process could be divided into four scenarios, as shown follows.

Case 1: α is odd, β is even.

In this case, the added group will still have an odd $(m+l)$ number of storage nodes which will not generate "bubbles" as the case of even number nodes do. So all added nodes are required to fulfilled with the objects.

According to the shifted declustering layout, the layout scenario after addition process is as follows.

$$q_1 = (\alpha + l - \beta)/2 \tag{A.1}$$

$$z_1 = \lfloor a/(\alpha + \beta) \rfloor \tag{A.2}$$

$$y_1 = (z_1 \% q_1) + 1 \tag{A.3}$$

$$\mathbf{disk}(a, i)_1 = (a + iy)\%(m + l) \tag{A.4}$$

In order to calculate the objects moved, we will compare $\mathbf{disk(a,i)}$ (before addition) and $\mathbf{disk(a,i)_1}$ (after addition) when a and i are the same in both case.

1. In the first iteration ($z_1 = 0$, $y_1 = 1$), $\mathbf{disk}(a,i)_1 = (a+i)\%(m+l)$ (after addition) while $\mathbf{disk}(a,i) = (a+i)\%m$ (before addition). When $(a+i) \leq m$, the objects will not moved. When $m < (a+i) \leq (m+l)$ the objects will be moved from the following iterations and three more objects are substituted in the front by the coming objects. *e.g.* in Figure 2.1 a), if $l = 2$, redundancy group (9,i) and (10,i)($i = 0, 1$ or 2) will be moved to the newly added nodes and (7,2), (8,1) and (8,2) are moved to the newly added nodes. So the number of moved objects is $3 * (\beta + 1)$.

2. In the following iterations ($z_1 \geq 1$,$y_1 \geq 1$), we can consider it as stuffing empty slots— each iterations' first $l * z$ slots will be empty because they are moved to the previous iterations. Moreover, since the total node number is odd and the empty slots' number are even, there is no chance for $\mathbf{disk_1(a,i)} = \mathbf{disk(a,i)}$. So every following objects in this iteration are needed to move to the previous slots and fill them. In this case the moved objects are $(q-1)\alpha * 3$.

Add up the above results, the total reorganized objects are $[(q-1)\alpha + (\beta+1)] \times 3$.

Case 2: α is odd, β is odd. In this case, the group number will change from odd to even, and "bubbles" are added to maintain the SD layout.

After addition, the layout equations are as follows.

$$q_2 = (\alpha + \beta - 2)/2 \tag{A.5}$$

$$z_2 = \lfloor a/\alpha + \beta - 2 \rfloor \tag{A.6}$$

$$y_2 = (z\%q') + 1 \tag{A.7}$$

$$d_{b2} = (\alpha + \beta - 1 - z)\%(m + l) \tag{A.8}$$

$$\mathbf{disk}(a,0)_2 = a\%(\alpha + l) \tag{A.9}$$

$$\mathbf{disk}(a,i)_2 = \begin{cases} (a + iy + 1)\%(\alpha + \beta)^{*} \\ (a + iy)\%(\alpha + \beta)^{**} \end{cases}, (i \geq 1) \tag{A.10}$$

*: if one of the following applies:

i) $\mathbf{disk}(a,0)_2 < d_{b2}$ and $\mathbf{disk}(a,0) + iy \geq d_{b2}$

ii) $\mathbf{disk}(a,0)_2 > d_{b2}$ and $\mathbf{disk}(a,0) + iy \geq d_{b2} + n$

where $\mathbf{disk}(a,i)_2$ is the disk position

**:otherwise

for object (a,i). Then we compare the $\mathbf{disk}(a,i)$ (before) and $\mathbf{disk}(a,i)_2$ (after) to determine how many objects are needed to be reorganized.

1. In the first iteration ($z_2 = 0$, $y_2 = 1$), $\mathbf{disk}(a,i)_2 = (a+i)\%(\alpha+\beta)$ and the "bubble" $d_b 2 = (\alpha + \beta - 1)$. While the original $\mathbf{disk}(a,i) = (a+i)\%\alpha$ and no "bubble" is in it. So normally the total number of objects moved are $3 * l$. Except one case: when $\beta = 1$, there is no need to change any object position, so the number of moved objects are 0 as Equation 4.1*** shows.

2. In the following iterations ($z_2 \geq 1$,$y_2 \geq 1$), no matter l is one or not, "bubble"s are generated in each iteration. The first z column of objects are moving after the

"bubble", which makes the reorganization process similar as Case 1—all the objects are reorganized to other nodes. The reorganization overhead is $(q-1)\alpha * 3$.

So the total number of reorganized objects is $(q-1)\alpha \times 3$ when $\beta = 1$ and $[(q-1)\alpha + \beta] \times 3$ when $\beta \leq 1$.

Case 3: α is even, β is odd. In this case, the group number is becoming to odd $(\alpha+\beta)$ after addition. Since there are "bubbles" in the layout before addition, they will be filled out after adding nodes into the group. After addition, the layout equations are as follows.

$$q_3 = (\alpha + \beta - 1)/2 \tag{A.11}$$

$$z_3 = \lfloor a/(m+l) \rfloor \tag{A.12}$$

$$y_3 = (z_3 \% q_3) + 1 \tag{A.13}$$

$$\mathbf{disk}(a,i)_3 = (a + iy)\%(m+l) \tag{A.14}$$

where $\mathbf{disk}(a,i)_3$ is the disk position for object (a,i). We compare the disk position $\mathbf{disk}(a,i)$ (before) and $\mathbf{disk}(a,i)_3$ (after)for the object (a,i).

1. In the first iteration ($z_3 = 0$, $y_3 = 1$), $\mathbf{disk}(a,i)_3 = (a+i)\%(\alpha+\beta)$ (after addition) while $\mathbf{disk}(a,i) = (a+i)\%m$ (before addition) and $d_b = (\alpha - 1)$. This means there are $3 * (\beta + 1)$ objects from the second iteration as Case 1 did and 3 extra objects to fill the "bubble". However, there is one exception: when $a = m-1$, $\mathbf{disk}(a,0) = \alpha - 1$ and $\mathbf{disk}(a,0)_3 = \alpha - 1$, this object (m-1,0) is moved from second iteration to the first iteration with a different offset but a same disk. So the total number of objects reorganized for the first iteration is $(\beta + 1) * 3 + 2$

2. In the following iterations ($z_3 \geq 1$,$y_3 \geq 1$), $\textbf{disk}(a,i)_3 = (a + iy_3)\%(\alpha + \beta)$ (after addition) while $\textbf{disk}(a,i) = (a + iy)\%m$or $(a + iy + 1)\%m$ (before addition). The situation is similar to Case 1, all objects are needed to be reorganized to a different disk location.

So the total number of reorganized objects is $[(q-1)\alpha + (\beta+1)] \times 3 + 2$.

Case 4: α is even, β is even. In this case, the group will still have even $(m+2)$ number of nodes. Adding nodes will not result in the fulfillment of the "bubbles". After addition, the layout equations are as follows.

$$q_4 = (\alpha + \beta - 2)/2 \tag{A.15}$$

$$z_4 = \lfloor a/\alpha + \beta - 2 \rfloor \tag{A.16}$$

$$y_4 = (z\%q') + 1 \tag{A.17}$$

$$d_{b4} = (\alpha + \beta - 1 - z)\%(\alpha + \beta) \tag{A.18}$$

$$\textbf{disk}(a,0)_4 = a\%(\alpha + \beta) \tag{A.19}$$

$$\textbf{disk}(a,i)_4 = \begin{cases} (a + iy + 1)\%(\alpha + \beta)^* \\ (a + iy)\%(\alpha + \beta)^{**} \end{cases}, (i \geq 1) \tag{A.20}$$

*: if one of the following applies:

i) $\mathbf{disk}_4(a,0) < d_{b4}$ and $\mathbf{disk}(a,0) + iy \geq d_{b4}$

ii) $\mathbf{disk}_4(a,0) > d_{b4}$ and $\mathbf{disk}(a,0) + iy \geq d_{b4} + \alpha$

where $\mathbf{disk}(a,i)_2$ is the disk position

*:otherwise

for object (a,i). Then we compare the $\mathbf{disk}(a,i)$ (before addition) and $\mathbf{disk}(a,i)_4$ (after addition) to determine how many objects are needed to be reorganized.

1. In the first iteration ($z_4 = 0$, $y_4 = 1$), $\mathbf{disk}(a,i)_4 = (a+i)\%(\alpha+\beta)$ and the "bubble" $d_{b4} = \alpha + \beta - 1$. While the original $\mathbf{disk}(a,i) = (a+i)\%(\alpha - 1)$ and the "bubble" $d_b = \alpha - 1$. So the total number of objects moved are $3 * (\beta + 1)$. Also, we have to the similar exception as the one in Case 3 Proof 1): when $a = \alpha - 1$, $\mathbf{disk}(a,0) = \mathbf{disk}(a,0)_4$. So the number of moved objects $3 * \beta + 2$.

2. In the following iterations ($z_2 \geq 1$,$y_2 \geq 1$), the reorganization process is similar to Case 1—all the objects are reorganized to other nodes. The reorganization overhead is $(q-1)\alpha * 3$.

So the total number of reorganized objects is $[(q-1)\alpha + \beta] \times 3 + 2$

LIST OF REFERENCES

[1TB07] "Seagate, Samsung Begin to Ship 1TB Desktop Hard Drives." http://www.dailytech.com/Article.aspx?newsid=7740, 2007.

[BO03] P. J. Braam and Others. "The Lustre storage architecture." *White Paper, Cluster File Systems, Inc., Oct*, **23**, 2003.

[Bor07] D. Borthakur. *The Hadoop Distributed File System: Architecture and Design.* The Apache Software Foundation, 2007.

[CF96] Peter F. Corbett and Dror G. Feitelson. "The Vesta parallel file system." *ACM Trans. Comput. Syst.*, **14**(3):225–264, 1996.

[CT96] Shenze Chen and Don Towsley. "A Performance Evaluation of RAID Architectures." *IEEE Trans. Comput.*, **45**(10):1116–1130, 1996.

[ear] "Earth Surface and Interior." `http://science.nasa.gov/earth-science/focus-areas/surface-and-interior/`.

[fac] "Facebook." `http://www.facebook.com/`.

[goo] "Google." `http://www.google.com/`.

[HD90] Hui-I Hsiao and David J. DeWitt. "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines." In *Proceedings of the Sixth International Conference on Data Engineering*, pp. 456–465, Washington, DC, USA, 1990. IEEE Computer Society.

[HM04] R.J. Honicky and E.L. Miller. "Replication under scalable hashing: a family of algorithms for scalable decentralized data distribution." *In Proceedings of 18th International Parallel and Distributed Processing Symposium, 2004.*, p. 96, 26-30 April 2004.

[HZJ10] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian. "Supporting Scalable and Adaptive Metadata Management in Ultra Large-scale File Systems." *IEEE Transactions on Parallel and Distributed Systems*, **99**(PrePrints), 2010.

[KLL97] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web." In *In ACM Symposium on Theory of Computing*, pp. 654–663, 1997.

[lln] " I/O Performance Experiments." `http://www.citi.umich.edu/projects/asci/\benchmarks.html`.

[LNS94] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. "RP*: A Family of Order Preserving Scalable Distributed Data Structures." In *VLDB*, pp. 342–353, 1994.

[LR02] W. Litwin and T. Risch. "LH*g: a high-availability scalable distributed data structure by record grouping." *Knowledge and Data Engineering, IEEE Transactions on*, **14**(4):923 –927, jul. 2002.

[MM02] P. Maymounkov and D. Mazires. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric." In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pp. 53–65. Springer Berlin / Heidelberg, 2002.

[pvf] "The PVFS Project." `http://www.pvfs.org/`.

[S 03] S.T. Leung S. Ghemawat, H. Gobioff. "The Google File System." *OPERATING SYSTEMS REVIEW*, **37**:29–43, 2003.

[SG07] Bianca Schroeder and Garth A. Gibson. "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?", 2007.

[slo] "The Sloan Digital Sky Survey." `http://www.sdss.org/`.

[SML03] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. "Chord: a scalable peer-to-peer lookup protocol for Internet applications." *Networking, IEEE/ACM Transactions on*, **11**(1):17 – 32, feb. 2003.

[SSD96] Adam Sweeney, Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. "Scalability in the XFS File System." In *In Proceedings of the 1996 USENIX Annual Technical Conference*, pp. 1–14, 1996.

[Vaj09] P. Vajgel. "Needle in a haystack:efficient storage of billions of photos." `http://www.facebook.com/note.php?note_id=76191543919`, 2009.

[VDF02] MA) Venkatesh, Dinesh (North Andover, MA) Duso, Wayne W. (Shrewsbury, MA) Forecast, John (Newton, and MA) Gupta, Uday (Westford. "Reorganization of striped data during file system expansion in a data storage system.", December 2002.

[WBM06a] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. "Ceph: A scalable, high-performance distributed file system." In *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI*, pp. 307–320, 2006.

[WBM06b] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. "CRUSH: controlled, scalable, decentralized placement of replicated data." In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, p. 122, New York, NY, USA, 2006. ACM.

[WWA93] Randolph Wang, Olph Y. Wang, and Thomas E. Anderson. "xFS: A Wide Area Mass Storage File System." In *In Fourth Workshop on Workstation Operating Systems*, pp. 71–78, 1993.

[XSM05] Qin Xin, Thomas J. E. Schwarz, and Ethan L. Miller. "Disk infant mortality in large storage systems." In *In Proc of MASCOTS 05*, pp. 125–134, 2005.

[yah] "Yahoo!" `http://www.yahoo.com/`.

[ZGW08] Huijun Zhu, Peng Gu, and Jun Wang. "Shifted declustering: a placement-ideal layout scheme for multi-way replication storage architecture." In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pp. 134–144, New York, NY, USA, 2008. ACM.